Argonne NATIONAL LABORATORY

specpy: a simplified approach for controlling beamlines in Python

Brian Toby

Lakhsmipriya Sukumar

(an extension of PyEpics, by Matt Newville, CARS)

U.S. DEPARTMENT OF ENERGY

Goal: create a convenient code library for scripting beamline actions in Python

Motivation: writing Python scripts for beamline automation can be time consuming, especially in comparison to spec, where libraries of scripts simplify many tasks.

- Library designed around needs of 1-ID, but most routines should be of general use.
  - Control motors, scans, scalers in a manner similar to spec
  - Reuse spec command names, variables, etc. where it makes sense
  - Simplify configuration
  - Allow novice Python programmers to write scripts without much knowledge of PyEpics, Matplotlib & wxPython
  - Simulation mode: make it easy to connect and disconnect the package from EPICS
    - Allow development and testing of scripts on computers without PyEpics or without access to beamline
    - Optionally test moves for motor soft limits, even without EPICS access
- Prerequisite: knowledge of Python

specpy: Where to find it and its documentation

- Home page: https://subversion.xray.aps.anl.gov/trac/spec1ID
- Download with subversion from:
  https://subversion.xray.aps.anl.gov/spec1ID/specpy/trunk/
- HTML documentation:
  https://subversion.xray.aps.anl.gov/spec1ID/specpy/trunk/docs/build/html/index.html
- PDF documentation (48 pages, contents same as HTML):
  https://subversion.xray.aps.anl.gov/spec1ID/specpy/trunk/docs/build/latex/EPICSPythonSPECmotor.pdf

Modules

- *spec: SPEC-like emulation Motor interface routines*
  - *Access motors and scalers*
- *macros: Additional SPEC-like emulation General Purpose Routines*
  - *Logging of PVs, etc, plotting, user input, monitoring of PVs*
- *AD: Area-Detector access Detector Access Routines*
  - *Control area detectors*
- *GE: GE Image processing Summary*
  - *Processing with GE raw data files*
- *s1id: 1-ID specific routines*
  - *Implementation specific to 1-ID*

Configuration Routines

The configuration routines are simple and are intended to be included in one or more setup files (perhaps one for users and another for beamline staff)

- `spec.DefineMtr('mtrXX1','ioc1:mtr98','(mm) +up, -10 to +10')`
  - Defines symbol `spec.mtrXX1` to use motor with PV root `ioc1:mtr98`. Optional description string informs operator about the motor.
- DefinePseudoMtr
  - Defines a "motor" that is mapped to movement of other motors. Also note routine `spec.MoveMultipleMtr` (mmv and ummv) which avoid unnecessary moves when pseudo-motors are moved in groups.
- `spec.DefineScaler('id1:scaler1',16)`
  - Defines a connection to a scaler with PV root `id1:scaler1`
  - More than one scaler can be used by defining an index (index=0 is used by default)

```
>>> from spec import *
>>> EnableEPICS()
>>> DefineMtr('mtr1','ioc1:mtr98')
>>> DefineMtr('mtr2','ioc1:mtr99')
>>> from spec import *
```

```
>>> mv(mtr1, 0.123)
```

Logging Configuration

Logging is used to designate values that should be recorded as data are taken. (In module macros*). These could also be collected in a configuration file.*

- init_logging(): Initializes the list of items to be reported
- add_logging...(): Used to indicate a PV, scaler, motor position, scaler or Python global variable.
  - Any number of items can be logged.
- Logging is done at every step in a ascan/dscan

- In scripts, use `write_logging_header()` to write a file header
- At each step use `write_logging_parameters()` to add a line of values.
  - Files are in .csv format (other formats could be implemented)

Logging example code

Setup Logging

```
>>> import macros
>>> macros.init_logging()
>>> GE_prefix = 'GE2:cam1:'
>>> macros.add_logging_PV('GE_fname',GE_prefix+"FileName",
...                        as_string=True)
>>> macros.add_logging_PV('GE_fnum',GE_prefix+"FileNumber")
>>> macros.add_logging_motor(spec.samX)
>>> macros.add_logging_scaler(9)
>>> macros.add_logging_Global('var S9','spec.S[9]')
>>> macros.add_logging_PV('p1Vs',"1idc:m64.RBV")
```

Scan with Data Logging

```
>>> macros.write_logging_header(logname)
>>> spec.umv(spec.mts_y,stY)
>>> for iLoop in range(nLoop):
>>>     spec.umvr(spec.mts_y,dY)
>>>     count_em(Nframe*tframe)
>>>     GE_expose(fname, Nframe, tframe)
>>>     wait_count()
>>>     get_counts()
>>>     macros.write_logging_parameters(logname)
>>> mac.beep_dac()
```

Spec-like commands (in module spec)

- move motor: mvr() & mv()
- move motor with wait: umvr() & umv()
- move multple motors: mmv() [w/wait ummv()]
- where is this motor?: wm()
- where are all motors?: wa()
- start and readout scaler after completion: ct()
- start scaler and return: count_em()
- wait for scaler to complete: wait_count()
- read scaler: get_counts()
- turn simulation mode on: onsim()
- turn simulation mode off: offsim()
- array of motor positions and last count values: *A[] and S[]*
- Delay for n seconds: sleep()

For counting against a monitor, use *SetMon()* to select a scaler channel.

Module macros

Module macros *provides more complex capabilities typically found in spec macros*

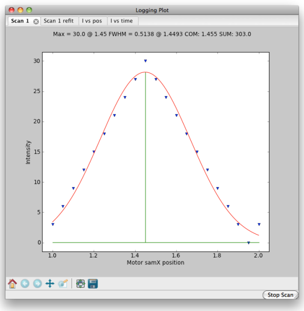
- Scans – scan a peak, plot and fit it
- Logging – designates parameters to be written to file and write them on routine call
- Plotting – designate parameters to be plotted and plot the current values on routine call
- Monitoring – designate parameters to be written into a file, triggered by a PV changing
- Get user input – prompted in terminal or from a GUI
- Send an e-mail (good with try/except blocks to respond to script errors)

scans

It is assumed that complex scans will be written with Python loops for greatest flexibility (except perhaps where speed is critical, when EPICS scan records are needed).

Routines `macros.ascan` *and* `macros.dscan` *provide peak finding etc.*

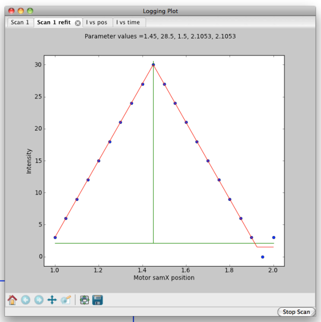
- `macros.ascan()` and `macros.dscan` ()
  - Provides plotting and Gaussian fitting.
  - Output file intended to be spec compatible
  - Any user-supplied function can also be fit using `macros.RefitLastSscan()`
- Use `spec.SetDet()` to determine which detector is plotted
- Use `macros.SetScanFile()` to set the file written during a `ascan/dscan`



```
Max = 30.0 @ 1.45 FWHM = 0.5138 @ 1.4493 COM: 1.455 SUM: 303.0
```

```
>>> macros.ascan(spec.samX,1,2,21,1,settle=.1)
```

Fitting a scan with an alternate function



Use `macros.RefitLastScan(`***`MyFit`***`)` to fit a user-defined function to the results from the last ascan or dscan.

To create ***`MyFit`*** derive a class from ***`FitClass`*** like this (see FitGauss and FitSawtooth in macros):

```
class MyFit() :
    def __init__(self,x,y):
        # define list of starting parameters
        # in terms of the obs x & y values
        self.startVals = [...]
    def Eval(self, parm, x):
        # evaluate the function for each x point
        # return as a list or np.array
        return [...]
```

Plotting in scripts

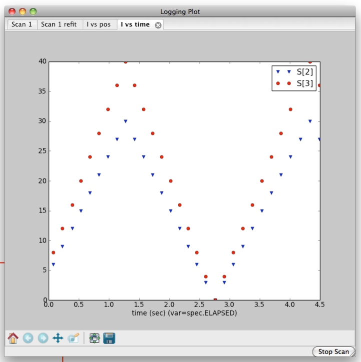
Designate parameters to be plotted in one call (`DefineLoggingPlot`) and plot the current values in another (`UpdateLoggingPlots`) call

- Plotting is setup by defining the plot to be generated using `DefineLoggingPlot()`

```
DefineLoggingPlot('I vs time',
    make_log_obj_Global('time',
                        'spec.ELAPSED'),
    make_log_obj_scaler(2),
    make_log_obj_scaler(3))
```

- Points are added to the plot using `UpdateLoggingPlots()`

```
spec.initElapsed()
spec.umv(spec.samX,2)
for iLoop in range(30):
    spec.umvr(spec.samX,0.05)
    spec.sleep(0.5)
    spec.ct(1)
    UpdateLoggingPlots()
```



Monitoring changes in PVs

The PyEpics module provides a mechanism for tracking changes to PVs. This is implemented in the specpy macros module with routines that plot or write to file values of EPICS PVs or other parameters in response to changes in EPICS PVs.

To filter the number of events that trigger a monitor response, one can set a "dead time" before another change should be logged and/or a specific value for the PV that triggers a response (recommended for integer values only).

Define PV to be monitored:

`DefMonitor(fileprefix, pv, monitorlist, pvvalue, delay)`

`fileprefix`	file name is fileprefix+timestamp.log
`pv`	PV to monitor for changes
`monitorlist`	list of PVs to list in log file
`pvvalue`	Log only if PV == pvvalue (optional)
`delay`	Log at most once per delay seconds (optional)

Monitoring is started when `StartAllMonitors()` is called [multiple `DefMonitor()` calls are allowed]

Plotting when PVs change

- Two PVs can be monitored, a change in one causes the plot to be cleared or a new plot tab to be started, the other triggers plotting. Can be used with EPICS scan record.

Define a plot using

```
SetupMonitorPlot(clearPV, plotPV, Xvar, Yvars, reusetab,
tablbl=None, clearPVvalue=None, cleardelay=None, plotPVvalue=None,
plotdelay=None)
```

`clearPV`	PV to trigger clearing of plot
`plotPV,`	PV to trigger plotting
`Xvar,`	PV to plot on x-axis
`Yvars`	List of PVs to plot on y-axis
`reusetab`	Clear plot (False) or create new tab (default)
`tablbl`	Label for plot tab (optional)
`clearPVvalue, cleardelay`	Set value and minimum interval for clear of plot (opt)
`plotPVvalue, plotdelay`	Set value and min interval for adding point to plot (opt)

Plotting is started when `StartAllMonitors()` is called [multiple `SetupMonitorPlot()` calls are allowed]

Generating an e-mail on an error

Best practice: write one routine to get input values & set up a measurement and a second to do the measurement (errors in input should not trigger an e-mail)

```
def DoSomething():
   parms = ...
   msg = 'Error in DoSomething'
   buglist = ['11bm@aps.anl.gov']
   def GetInput():
      <input code here>
   def DoMeasurement():
      <measurement code here>
   GetInput()
   try:
      DoMeasurement()
   except:
      macros.SendTextEmail(buglist, msg,
         subject='oops')
```

Routines to Get User Input

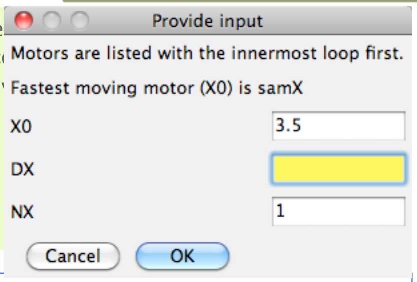
By following a simple protocol, input can be obtained from the user in a terminal window or in a GUI by setting one flag

Protocol:

- Use macros.SetGUImode(UseGUI) to set mode
- Call macros.setupInput()
- Call macros.ShowLine() to show user prompts lines
- Create an empty list for values from user
- Call macros.UserIn() for each value from user (specify a prompt, an initial value or None, a data type and the name of the list for values
- Call macros.finishInput() to get the input

Input Example

```
Motors are
Press ^D (c
Fastest mo
X0 (3.5):
DX (None):
DX (None):
NX (1):
X0, DX, NX
```



Motors are listed with the innermost loop first.

Fastest moving motor (X0) is samX

X0	3.5
DX	
NX	1

This code will obtain three values fro
is True.

```
macros.SetGUImode(UseGUI)
macros.setupInput()
macros.ShowLine("Motors are listed with the innermost loop first.")
if not macros.GetGUImode():
    macros.ShowLine("Press ^D (control D) to abort input")
macros.ShowLine("motor (X0) is "+str(spec.GetMtrInfo(spec.m3)
['symbol']))
result = []
macros.UserIn("X0",spec.wm(spec.m3),float, result)
macros.UserIn("DX",None,float, result)
macros.UserIn("NX",1,int, result)
if not macros.finishInput(): return
X0, DX, NX = result
print 'X0, DX, NX = ', X0, DX, NX
```

Area Detector Interface

The EPICS AreaDetector module defines a common interface for area detectors – sort of. In truth there are frequently quite different parameters that must be set for each type of detector. The specpy AD module defines a framework that makes it easier to abstract detector actions. Detectors are defined in two stages:

1. Define a detector with `AD.DefineAreaDetector()`
   – Specify a symbolic name for the detector, a symbolic detector type and one or two PV prefixes:

   ```
   DefineAreaDetector('GE4', 'GE', 'GE4:cam1')
   DefineAreaDetector('ScintX','ScintX','ScintX:cam1','ScintX:TIFF1:')
   ```

2. Define commands to be used with a set of detectors types with `AD.defADcmd()`

   ```
   defADcmd('trigger_mode','%TriggerMode','%TriggerMode_RBV',
       'Trig. mode', det='ScintX', enum=(0,1))
   defADcmd('frames', '%NumImages', '%NumImages_RBV', '# of frames',
      enum='0 < val < 300')
   ```

Note that '%' or '%%' specifies the first or second PV prefix, enum specifies a tuple of allowed values or an expression to use to check validity.

Commands to use with Area Detectors

- `AD_get()` Read an area detector parameter
- `AD_set()` Set an area detector parameter
- `AD_done()` Test if the detector(s) have completed data collection
- `AD_show()` Shows defined commands options for `AD_get()` and `AD_set()` for a detector type

- `AD_acquire()` Set the filename, count time and frames and collect
  - Combines several `AD_set()` and `AD_get()` operations

The AD module will probably need more development for use at other sectors

1-ID specific modules

- Module GE:
  - Reads files from GE, plotting images, computing average intensities for regions of interest (ROI), plots ROI values
  - Future work: incorporate data reduction steps

- Module s1id:
  - Functions that access specific PVs for 1-ID, such as opening hutch shutters, enabling the fast shutter, recording motor soft limits
  - These functions are worth looking at as examples, since they attempt to operate in a fail-safe way. Operations are tried multiple times and results are tested and errors (exceptions) are raised, so that a script will fail rather than run because a shutter fails to open on the first try.